Frontend Performance Optimization - Complete guide

1. Performance Monitoring

Performance monitoring allows you to proactively detect, investigate, and resolve performance issues. It's essential to collect both real-user data (RUM) and lab data. RUM reflects real-world performance, while lab tools simulate load scenarios to uncover bottlenecks.

- **FCP (First Contentful Paint):** Marks the time when the browser renders the first piece of DOM content.
- **LCP (Largest Contentful Paint):** Measures load speed by reporting the render time of the largest content block.
- **FID (First Input Delay):** Time between a user's interaction and the browser's response.
- **CLS (Cumulative Layout Shift):** Measures visual stability by tracking layout movements.
- **TTI (Time to Interactive):** Indicates when a page becomes fully interactive.

Track metrics using `PerformanceObserver`, Google Analytics, or third-party tools like Sentry, Datadog, or New Relic.

2. Performance Tools

Tools provide critical insights into frontend performance and user experience:

- **Chrome DevTools:** Inspect network activity, JavaScript execution, layout recalculations, and repaint timings.
- **Lighthouse:** Performs audits on load speed, accessibility, best practices, SEO, and PWA readiness. Use it via Chrome or CLI.
- **WebPageTest:** Offers deep visibility into multi-step transactions, time to first byte, connection reuse, and waterfall charts.
- **GTMetrix:** Combines Google and YSlow recommendations with historical performance tracking.
- **Core Web Vitals Extension:** Offers real-time CWV reporting in your browser during development.

3. Network Optimization

Optimizing how resources are fetched and delivered directly improves time to load and perceived performance:

- **Compression:** Enable Brotli or Gzip on the server to reduce asset size.
- **Caching:** Set HTTP cache headers (e.g., Cache-Control, ETag) to store static assets on the client or CDN.
- **Prefetch/Preload:** Use `<link rel="preload">` and `<link rel="dns-prefetch">` to reduce wait times.
- **Lazy loading:** Defer loading of non-critical images and components using `loading="lazy"`.
- **Minify assets:** Remove unnecessary whitespace and comments from CSS, JS, and HTML.
- **Use a CDN:** Reduce latency by serving assets from geographically distributed servers.

4. Rendering Patterns

Rendering strategies dictate how content reaches the user's screen. Choosing the right one depends on your use case:

Client-Side Rendering (CSR)

Entire HTML is rendered using JavaScript after the page is loaded. Initial load may be slow but great for dynamic SPAs.

Server-Side Rendering (SSR)

HTML is generated on the server on each request. Better for SEO and initial load times. Requires backend rendering logic.

```
// Angular Universal SSR example
// Step 1: Add Angular Universal
ng add @nguniversal/express-engine

// Step 2: Build and serve SSR app
npm run build:ssr && npm run serve:ssr

// Angular CLI will create server.ts for Node.js rendering
```

Static Site Generation (SSG)

HTML is pre-built at deploy time. Best for documentation, blogs, and sites with infrequent updates. Offers extremely fast performance.

Incremental Static Regeneration (ISR)

Hybrid approach allowing on-demand static generation while keeping previously cached content. Supported in frameworks like Next.js.

5. Build Optimization

Optimizing your frontend build pipeline ensures only the essential code reaches the browser:

- **Code Splitting:** Dynamically load modules as needed instead of bundling all at once.
- **Tree Shaking:** Use ES module syntax to eliminate unused code during build.
- **Minification:** Shrink your JavaScript and CSS using tools like Terser, UglifyJS, or CSSNano.
- **Image Optimization:** Convert images to WebP/AVIF, use responsive images, and compress via tools like Squoosh.
- **Long-term Caching:** Use content-hashed filenames and configure service workers or cache manifest.
- **Bundle Analysis:** Use Webpack Bundle Analyzer or Source Map Explorer to visualize and reduce bundle size.
- **Environment Configurations:** Strip dev-only logic using `process.env.NODE_ENV` checks.

```
// Angular Lazy-loaded module example
// app-routing.module.ts
const routes: Routes = [
  { path: 'dashboard', loadChildren: () => import('./dashboard/dashboard.module'
];
```

```
// Angular Environment-specific optimization (in angular.json)
"configurations": {
  "production": {
    "optimization": true,
    "outputHashing": "all",
    "sourceMap": false,
    "extractCss": true
```

```
  }
}
```

Summary Table

Category	Key Focus	Example Technique
Performance Monitoring	Real-time UX metrics	CLS, LCP, FID tracking with RUM
Performance Tools	Analysis & auditing	Lighthouse, Chrome DevTools
Network Optimization	Reduce asset delivery time	Gzip, lazy loading, CDN
Rendering Patterns	Content delivery strategy	CSR, SSR, SSG, ISR
Build Optimization	Minimize shipped code	Code splitting, tree shaking